IMAC 1

Franck TRAN
Laurent BARBAT

2e semestre 2004

Projet MOSAIQUE



Création d'image panoramique

Mathématiques pour l'Informatique

Enseignent : Monsieur Nozick

Institut Gaspard Monge
Institut Charles Cros
Université de Marne la Vallée, pôle Val d'Europe

SOMMAIRE

INTRODUCTION ---------- p.02

I - **Mathématiques**
Explication des méthodes mathématiques :
- pivot de Gauss total. ---------- p.03
- multiplication de matrices ---------- p.04
- matrice transposée ---------- p.04
- inversion de matrices ---------- p.04

II – **Pseudo code et explication des principaux algorithmes**
Algorithmes mathématiques
- Pivot de gauss ---------- p.05
- Systèmes surdéterminés ---------- p.07
Homographies ---------- p.07
- Homographie Directe ---------- p.09
- Homographie Inverse ---------- p.10
Option
- Anti-aliasing ---------- p.10

III – **Structures de données**
Matrice ---------- p.11
Pixel / pixDouble
RVB
Image

IV – **IHM – OpenGL**
Description ---------- p.12
Difficultés

V – **Manuel d'utilisation** ---------- p.12

VI – **Exemple d'utilisation** ---------- p.13

VII – **Difficultés rencontrées** ---------- p.13

VIII – **Compte rendu** ---------- p.14

CONCLUSION ---------- p.15

ANNEXES ---------- p.16

INTRODUCTION

Ce projet de Mathématique pour l'Informatique a été réalisé dans le cadre de notre première année de formation d'Ingénieur IMAC.

Il nous arrive parfois de prendre plusieurs photographies à la suite pour effectuer un panorama. Cependant, le résultat n'est, la plupart du temps, celui escompté ; les raccords entre les photos ne 'collent' pas.

Le projet 'Mosaïque' est un programme permettant de corriger ces défauts, grâce à l'utilisation du pivot de Gauss et à l'application d'homographies aux images.

Ce rapport a pour but de donner une vue d'ensemble de notre projet, il contient aussi bien l'explication les différentes méthodes mathématiques abordées, que des algorithmes qui ont été implémentés dans notre programme. Nous avons ajoutés également des idées d'amélioration qui n'ont pas été codées car nous n'avons pas eu le temps le temps de le faire.

I - Mathématique

Ce projet comporte une grosse partie mathématique qui se traduise par la conception d'une bibliothèque mathématique spécifique au besoin du projet. Cette bibliothèque comporte quelques opérations de base sur les matrices dont nous allons vous présenter.

PIVOT DE GAUSS TOTAL

Le pivot de Gauss est une méthode de résolution de systèmes d'équation linéaire c'est la plus simple et la plus classique :

- De la première équation, on exprime la première inconnue $x1$ en fonction des autres inconnues, puis on substitue cette expression dans les autres équations du système, faisant apparaître ainsi un sous-système dans lequel $x1$ n'intervient pas,

· On itère le même schéma jusqu'à ce que la matrice du système devienne triangulaire supérieure (toute la partie inférieure de la matrice – sous la diagonale – est remplie de zéros),

· Une fois la triangularisation terminée, on calcule xn, puis xn-1, et ainsi de suite jusqu'à $x1$; les différentes solutions sont ainsi calculées de proche en proche.

Le principe de triangularisation permet ainsi de transformer le système initial

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & & \vdots \\ \vdots & & \ddots & \vdots \\ a_{n1} & \cdots & \cdots & a_{nn} \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

par un système triangulaire équivalent de la forme

$$\begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1n} \\ 0 & a'_{22} & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a'_{nn} \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_n \end{pmatrix}$$

Pour éliminer $x1$ de la première équation, on divise par le coefficient $a11$ de la matrice ; ce coefficient s'appelle le pivot. Il est clairement indispensable que ce coefficient ne soit pas nul.

Néanmoins, ce critère, parfaitement acceptable mathématiquement, n'est pas suffisant numériquement. En effet, ce coefficient peut être très faible, entraînant ainsi l'apparition de très grandes valeurs, et donc de grands risques d'imprécisions et d'erreurs numériques.

En fait, la méthode du pivot de Gauss est rigoureuse mathématiquement, mais elle conduit généralement à de nombreux calculs ; la méthode est donc assez sensible aux erreurs numériques, en particulier pour les systèmes de grande taille.

Une implémentation de cette méthode doit donc impérativement mettre en oeuvre une stratégie de choix du pivot, consistant à recherche le plus grand pivot en valeur absolue. On a pour cela plusieurs possibilités :

- la première méthode, la plus simple, consiste à observer que l'on peut indifféremment inverser l'ordre des équations, sans changer le système ni sa solution ; on recherche donc le pivot défini par $max_{l \leq i \leq n} |a_{i1}|$ et on permute les équations si nécessaire : **la recherche du plus grand pivot se fait donc par colonne**.

- la deuxième méthode consiste à **rechercher le plus grand pivot par ligne** plutôt que par colonne ; on recherche donc le pivot défini par $max_{l \leq i \leq n} |a_{1i}|$: l'inconvénient majeur de cette méthode est qu'elle ne conserve pas le système initial, mais qu'il faut obligatoirement tenir compte d'un réagencement des solutions ; cette méthode est donc plus compliquée que la précédente puisque le programme doit mémoriser toute la séquence de réagencements successifs afin de pouvoir remettre les solutions dans le bon ordre.

- la troisième méthode consiste à mixer les deux méthodes précédentes, et donc à chercher le pivot défini par $max_{l \leq i \leq n} (|a_{li}|,|a_{il}|)$; **la recherche se fait donc sur lignes et colonnes** : cette méthode a généralement les mêmes inconvénients que la précédente.
C'est cette dernière méthode que nous avons choisi d'implémenter dans notre programme, c'est le **pivot de Gauss total**

MULTIPLICATION DE MATRICES

colonne k

$$\text{ligne } i \rightarrow \begin{pmatrix} & & a_{1j} & & \\ & & \vdots & & \\ a_{i1} & \cdots & a_{ij} & \cdots & a_{in} \\ & & & & \\ & & a_{mj} & & \end{pmatrix} x \begin{pmatrix} & & b_{1k} & & \\ & & \vdots & & \\ b_{j1} & \cdots & b_{jk} & \cdots & b_{jp} \\ & & \vdots & & \\ & & b_{nk} & & \end{pmatrix} = \begin{pmatrix} & & \vdots & & \\ & & \vdots & & \\ \cdots & \cdots & c_{jk} & \cdots & \cdots \\ & & \vdots & & \\ & & \vdots & & \end{pmatrix} \leftarrow \text{ligne } i$$

colonne k

$$\underline{A} \quad x \quad \underline{B} \quad = \quad \underline{C}$$

	A	B	C
Ligne	1≤i≤m	1≤j≤n	1≤i≤m
Colonne	1≤j≤n	1≤k≤p	1≤k≤p

L'élément $C_{ik,}$ de la colonne k et de la ligne i de la matrice C, est la somme des n produits $a_{i1}.b_{1k}$,……, $a_{in}.b_{nk}$ des éléments de la ligne i de la matrice A par des éléments du même rang de la colonne k de la matrice B. La multiplication des matrices se fait lignes par colonnes.

MATRICE TRANSPOSEE

On appelle transposée d'une matrice A, la matrice notée A^T, obtenu à partir de A en échangeant les lignes et les colonnes

$$\underline{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & \cdots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & \cdots & a_{mn} \end{pmatrix} \qquad \underline{A}^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & \cdots & a_{m2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{1n} & a_{m2} & \cdots & \cdots & a_{mn} \end{pmatrix}$$

INVERSION DE MATRICES

On dit qu'une matrice A est inversible si il existe une matrice A^{-1} tel que A. A^{-1} = Id. Pour obtenir la matrice A^{-1}, il suffit de faire une résolution de système d'équation avec la méthode du pivot de Gauss en prenant une matrice A et une matrice identité Id.

II – Pseudo-code et explication des principaux algorithmes.

A – Algorithmes mathématiques

PIVOT DE GAUSS

La recherche du pivot de Gauss s'effectue en plusieurs étapes dont les principaux algorithmes se trouvent dans *maths.c* . Cette méthode permet de résoudre des équations dont le résultat est une matrice à plusieurs dimensions (et pas seulement un vecteur).

Nous recevons en paramètres les pointeurs a, b et xIn (xIn est vide, il sert à sauvegarder la matrice résultat).

Nous utilisons une matrice X (de la même taille que xIn) pour garder en mémoire l'indice initial des lignes qui seront certainement permutées lors des recherches de pivots. X nous permettra de réordonner les éléments du vecteur résultat.

Nous **recopions les matrice a et b** dans des nouvelles matrices A et B, car elles vont être modifier.

Puis nous parcourons sous-matrice par sous-matrice la matrice A en recherchant le pivot (de chaque sous-matrice courante) par l'appel de la fonction trouvePivot :

La fonction de recherche du pivot parcoure une matrice à partir d'une ligne et d'une colonne. On utilise une variable *max* pour stocker la valeur maximale trouvée. Lorsqu'on trouve un élément dont la valeur absolue est supérieure à celle actuellement sauvegardée, il la remplace et *x* et *y* prennent alors les nouvelles coordonnées.

Les coordonnées sont directement modifiées en mémoire par les pointeurs *x* et *y* passés en paramètres.

Par *x* et *y*, nous connaissons les coordonnées du pivot ; on compare ces coordonnées avec celles du de l'élément (0,0) - c'est-à-dire *(pivot,pivot)* - de notre sous matrice.

Si elles sont différentes, c'est que le pivot de ne trouve pas sur la même ligne (pour *y*) et/ou sur la même colonne (pour *x*). Il faut donc les **permuter** :

- Si *pivot* est différent de *y*, nous permutons les lignes d'indices *pivot* et *y* des matrice *A* et *B.*
- Si *pivot* est différent de *x*, nous permutons les colonnes 'numéro' *pivot* et *x*, ainsi que les lignes *pivot* et *x* de la matrice X.

Puis vient la **triangularisation** progressive de la matrice A : à chaque passage dans la boucle, une nouvelle sous-matrice est traitée.

On cherche, pour chaque ligne après celle du pivot, le coefficient qui va servir à calculer la nouvelle sous-matrice.

On met à 0 la valeur qui se trouve juste en dessous le pivot. Nous calculons le reste de la ligne de la matrice A et pour la matrice B grâce à la formule :
(élément de la colonne courante) – (élément de la ligne du pivot et de la colonne courante) * coeff

Nous terminons la boucle par un test permettant d'en sortir au cas ou la valeur absolue du dernier pivot serait inférieur à EPS, EPS étant une variable definie en début de programme à 10^-8, ce qui provoquerait des erreurs dans le programme.

Pseudo-Code

```
 pivotGaussTot(Matrice *a, Matrice *b, Matrice *xIn) {
   int pivot, indpivot, ligne, colonne;
   int x,y;
   int i;
   int colB;
   double coeff;


   //Allouer les matrices intermédiaires
   A = allouerMat(a->sizeX, a->sizeY);
   B = allouerMat(b->sizeX, b->sizeY);
   X = allouerMat(1,a->sizeY);


   initialiserMatrice(xIn);

   Remplir les indices de X de 0 à sizeY;

   recopierMatrice a dans A;
   recopierMatrice b dans B;

   pour pivot allant de 0 à A->sizeY avec un pas de 1 {
         trouverPivot(A,pivot,pivot,&x,&y);

          si (pivot != y) {
             permuterLignes(A,pivot,y);
             permuterLignes(B,pivot,y);
          }

          si (pivot != x) {
             permuterColonnes(A,pivot,x);
             permuterLignes(X,pivot,x);
          }

          Triangulariser(A);

   }

    //Calcul en remontant
    pour colB allant de 0 à B->sizeX avec un pas de 1 {
              pour ligne allant de (A->sizeY)-1 à 0 avec un pas de 1 {
                       coeff = lireEltMat(B,colB,ligne);

                       pour colonne allant de ligne+1 à A->sizeX avec un pas de 1 {
                              coeff -= (lireEltMat(A,colonne,ligne) *
                     lireEltMat(xIn,colB,lireEltMat(X,0,colonne)));
                       }
                        ecrireEltMat(xIn,colB,lireEltMat(X,0,ligne),(coeff/lireEltMat(A,ligne,ligne)));
               }
      }
}
```

PSEUDO INVERSE (système surdéterminé)

```
Matrice* pseudoInvMat(Matrice *A, Matrice *B) {
	Matrice *matInv;
	Matrice *matPseudo;
	Matrice *tmp;
	Matrice *X ;

	tmp = multiplierMat(transposée(A),A);
	matInv = inverserMat(tmp);
	matPseudo = multiplierMat(matInv,transposéeMat(A));
	X = multiplierMat(matPseudo,B);

	retourner X;
}
```

B - Homographies

Voici tout d'abord une explication du déroulement du programme pour le calcul des homographies. Viendra ensuite, le pseudo code des fonctions d'homographie.

L'utilisateur définie les points de corrélation entre l'image 1 et 2, puis 2 et 3. Il en résulte deux listes *correlation_12* et *correlation_32*. Lorsqu'il appuie sur la touche *Entrée*, le traitement homographique peut commencer.

Le programme récupère les 2 listes (fonction *recupListe*), il vérifie leur validité par la fonction *nbClick* (2 appels de fonction, un pour chaque liste) : Si le nombre de points cliqués entre la première image et la seconde diffère, ou s'il y a moins de 4 points cliqués dans une image, le programme s'arrête en spécifiant l'erreur.

Grâce aux 2 listes, le programme va pouvoir créer la matrice A & B, puis **calculer les 2 matrices H d'homographie** :

La fonction *calcMat* créée les matrice A et B 'manuellement' puis détermine la méthode à utiliser pour calculer la matrice H, pivot de gauss si 4 paires de points sont cliquées, pseudo inverse si plus.

Pour la description du **pivot de gauss**, se référer au chapitre concerné, plus haut.

Un système est dit 'surdéterminé' dès lors que pour une inconnue, plusieurs solutions sont possibles.

Par exemple : 2a = 5
a=3

La méthode de la **pseudo inverse** permet de résoudre le système surdéterminé A . X = B, où X est l'inconnu, grâce à la formule de l'énoncé.

Nous procédons par étapes en utilisant les fonctions de transposition de matrice *transpoMat*, inversion de matrice *invMat* et multiplication de matrices *multiplierMat*.

Le vecteur renvoyé est de norme 8 qu'il convient de convertir grâce à *convVectToMat3_3* en matrice 3*3, sachant que le dernier et 9e élément vaut 1. Nous utilisons le fait qu'une matrice en mémoire est représentée comme un vecteur, c'est-à-dire de manière linéaire, pour utiliser la fonction *memcpy*.

Ces opérations sont répétées pour les 2 matrices d'homographie.

Nous allouons en mémoire la place nécessaire pour les 2 images résultats à afficher *imgRes* & *imgResInv*. Il s'agit de trouver les coordonnées extrêmes de l'image finale (= les 4 coins de l'image). Les coordonnées de départ sont celles de l'image 2 – image référence – car elle sera projetée directement dans le nouveau repère, sans déformation.

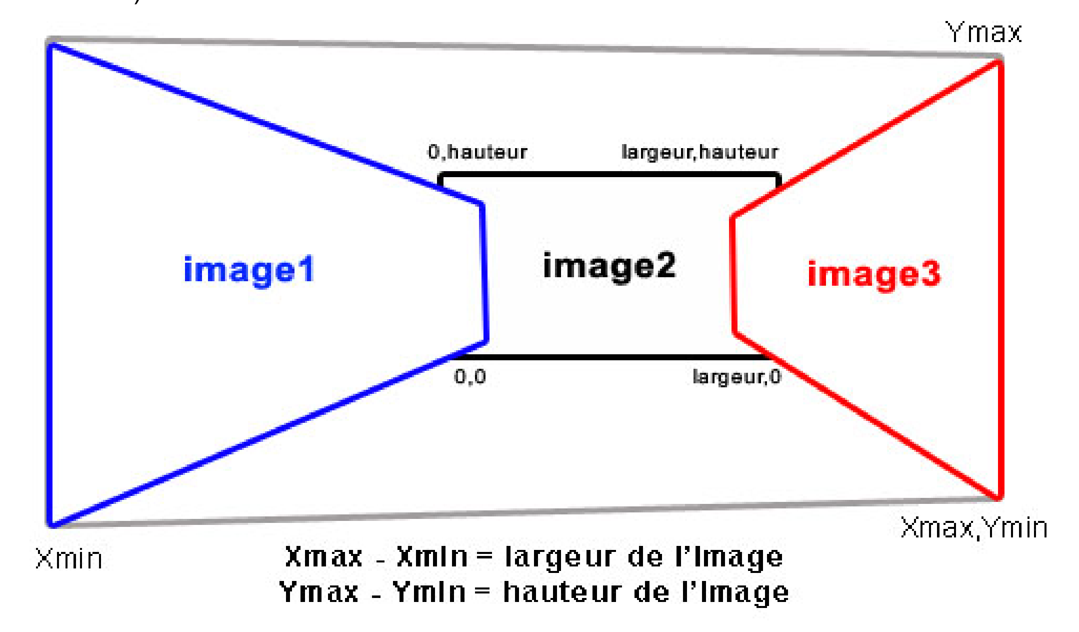
L'opération suivante va être appliquer 8 fois : 4 fois pour l'image 1 et 4 fois pour l'image 3. Chacune des 4 opérations correspond à un point extrême de l'image en question après projection sur le nouveau repère.

C'est donc pas à pas que nous allons trouver les **4 coordonnées de l'image finale** :

Appel de la fonction *homographiePoint* : calcule les coordonnées d'un point après projection.

Cette fonction reçoit la matrice d'homographie, les coordonnées *(x,y)* d'un des 4 points de l'image à projeter. Elle créée un vecteur de type *(x,y,1)*, puis multiplie ce vecteur par la matrice d'homographie. Sont renvoyés les coordonnées *(x,y)* du résultat de l'opération.

S'en suivent 4 tests destinés à savoir si ce point, ou pixel, fait partie de l'espace virtuel caractérisé par les 4 points extrêmes courants. Si oui, rien ne se passe ; sinon, ses coordonnées remplacent celles de l'un des 4 points de l'espace (*Xmin, Xmax, Ymin, Ymax sont les variables de type pixel* utilisées).



A la suite de cette opération répétée 8 fois, nous allons pouvoir créer l'image grâce à la fonction *newImage* qui créée en mémoire l'image aux dimensions spécifiées, et *InitImgNoir*, qui colorie l'image en noir en mettant toutes les composantes couleur des pixels de l'image à 0.

Nous **projetons l'image 2** sans aucun traitement sur le nouveau repère.

Nous parcourons l'image 2 et écrivons dans la nouvelle image tous les pixels en soustrayant la valeur minimale en abscisse à la valeur du pixel horizontal courant, et en soustrayant la valeur minimale en ordonnée à la valeur du pixel vertical. Ceci nous permet de replacer l'image 2 au centre de l'image finale.

Les valeurs minimales en hauteur et en largeur étant négatives, cela permet de décaler l'image vers la droite.

Ensuite, nous **devons projeter les images 1 et 3 sur le repère** grâce à *projectionImgDir*.

Nous agissons de la même façon que pour l'image. Seul ajout : la projection de chaque pixel via la fonction *homographiePoint* décrite plus haut.

Cette fonction est elle aussi appelée 2 fois, chacune avec une matrice homographique correspondant à l'image projetée.

Les résultats par homographie simple sont de manière général peut convaincant, du fait de la création d'une image plus grande que celles de départ : les pixels de l'image finale non pas toujours une correspondance avec ceux des images de départ (sauf l'image de référence).

La solution est d'utiliser **l'homographie inverse**, et donc de calculer X = H^{-1} . X' .

A partir de l'image finale allouée, le programme recherche une correspondance avec les 2 images de départ par la fonction *homographiePoint*, à partir des matrices d'homographie inversées. Si le point fait partie du 'cadre' de l'image 1 ou 3, il prend la couleur du pixel correspondant ; sinon, il reste noir.

Nous affichons les 2 images finales, résultat par homographie et homographie inverse grâce à la fonction glut *afficherImgFin*.

HOMORAPHIE DIRECTE

```
// fonction de projection d'une image sur un nouveau plan
void projectionImgDir(image *imgRes, image *img,Matrice *H, int Xmin, int Ymin) {
	int i,j ;
	pixel *proj;
	RVB *couleurs;

	Pour j allant de 0 à img->height
	Pour i allant de 0 à img->width
				proj = homographiePoint(H,i,j);
				couleurs = lirePixel(img,i,j);
				ecrirePixel(imgRes,(proj->x)-Xmin,
								(proj->y)-Ymin,couleurs);
		fin pour
	fin pour
}
pixel* homographiePoint(Matrice *H, int x, int y) {
	Matrice *matX;
	Matrice *matProj;
	pixel *res;
	int i;
	double w;

	matX = allouerMat(1,3);

	//Points de départ 	ecrireEltMatrice(matrice,colonne,ligne,valeur);
	ecrireEltMatrice(matX,0,0,x);
	ecrireEltMatrice(matX,0,1,y);
	ecrireEltMatrice(matX,0,2,1);

	matProj = multiplierMat(H,matX);

	//Calculs de coordonnees 2D (x'/w',y'/w',w'/w')
	w = matProj->data[2];
	pour i allant de 0 à 2 avec un pas de 1
		matProj->data[i] = (matProj->data[i]) / w;
	fin pour

	//Stocker dans pixel
	res->x = E ( matProj->data[0] );
	res->y = E ( matProj->data[1] );

	retourner res;
}
```

HOMOGRAPHIE INVERSE

```
void homographieInverse(image* img1, image* img3, image* imgRes, Matrice* H12, Matrice* H32, int
Xmin, int Ymin) {

	int i,j;
	Matrice* invH12 ;
	Matrice* invH32 ;
	pixel *pix ;
	RVB* pixColor ;

	allouerTailleImageFinale(pixcolor) ;

	invH12 = inverserMat(H12);
	invH32 = inverserMat(H32);

	pour i allant de 0 à imgRes->height avec un pas de 1
		pour j allant de 0 à imgRes->width  avec un pas de 1

			pix = homographiePoint(invH12,j+Xmin,i+Ymin);

			si  (pix->x >= 0 && pix->x < img1->width && pix->y >= 0
			&& pix->y < img1->height)
				pixColor = lirePixel(img1,pix->x,pix->y);
				ecrirePixel(imgRes,j,i,pixColor);
			fin si


			pix = homographiePoint(invH32,j+Xmin,i+Ymin);

			si  (pix->x >= 0 && pix->x < img1->width && pix->y >= 0
			&&  pix->y < img1->height)
				pixColor = lirePixel(img3,pix->x,pix->y);
				ecrirePixel(imgRes,j,i,pixColor);
			fin si
		fin pour
	fin pour
} // fin homographieInverse
```

C – Option

L'ANTI-ALIASING

L'anti-aliasing a pour but de corriger les problèmes « d'effet escalier ». En effet, lors du calcul de la position du pixel projeté, on est obligé de « caster » ce dernier en valeur entière afin de l'insérer à une position spécifique de l'image résultat. Cette façon de faire n'est pas très juste car on ne prend pas en compte cette différence de valeurs lors du « caste ». L'option anti-aliasing permet de faire un calcul pondéré de la valeur du pixel de couleurs avec les quatre pixels qui l'entourent en ayant un « poids » spécifique de chacun d'entre eux

Avec alpha la valeur ente 0 et 1, résultat de la soustraction de la valeur du pixel sur l'image finale par sa valeur entière (calcul réalisé lors de l'homographie, et seulement si l'option de l'anti aliasing '–a' est utilisé).

```
RVB* antiAliasing(double alpha, double beta, PixDouble* pixDouble, int i, int j, image* img) {
	RVB* poids, * color1, * color2, * color3, * color4;

	color1 = lirePixel(img, E(pixDouble->x) , E(pixDouble->y) -1);
	color2 = lirePixel(img, E(pixDouble->x) , E(pixDouble->y) +1);
```

```
	color3 = lirePixel(img, E(pixDouble->x) +1 , E(pixDouble->y) );
	color4 = lirePixel(img, E(pixDouble->x) -1 , E(pixDouble->y) );

	si (alpha < 0.5 && beta < 0.5){
		poids->r =	0.5*(alpha*color3->r+(1-alpha)*color4->r) +
				0.5*(beta*color1->r+(1-beta)*color2->r);
		poids->v =	0.5*(alpha*color3->v+(1-alpha)*color4->v) +
				0.5*(beta*color1->v+(1-beta)*color2->v);
		poids->b =	0.5*(alpha*color3->b+(1-alpha)*color4->b) +
				0.5*(beta*color1->b+(1-beta)*color2->b);
	}
	sinon si (alpha < 0.5 && beta > 0.5){
		poids->r =	0.5*(alpha*color3->r+(1-alpha)*color4->r) +
				0.5*(beta*color1->r+(1-beta)*color2->r);
		poids->v =	0.5*(alpha*color3->v+(1-alpha)*color4->v) +
				0.5*(beta*color1->v+(1-beta)*color2->v);
		poids->b =	0.5*(alpha*color3->b+(1-alpha)*color4->b) +
				0.5*(beta*color1->b+(1-beta)*color2->b);
	}
	sinon si (alpha > 0.5 && beta > 0.5) {
		poids->r =	0.5*(alpha*color3->r+(1-alpha)*color4->r) +
				0.5*(beta*color1->r+(1-beta)*color2->r);
		poids->v =	0.5*(alpha*color3->v+(1-alpha)*color4->v) +
				0.5*(beta*color1->v+(1-beta)*color2->v);
		poids->b =	0.5*(alpha*color3->b+(1-alpha)*color4->b) +
				0.5*(beta*color1->b+(1-beta)*color2->b);
	}
	sinon si (alpha > 0.5 && beta < 0.5) {
		poids->r =	0.5*(alpha*color3->r+(1-alpha)*color4->r) +
				0.5*(beta*color1->r+(1-beta)*color2->r);
		poids->v =	0.5*(alpha*color3->v+(1-alpha)*color4->v) +
				0.5*(beta*color1->v+(1-beta)*color2->v);
		poids->b =	0.5*(alpha*color3->b+(1-alpha)*color4->b) +
				0.5*(beta*color1->b+(1-beta)*color2->b);
	}
	retourner poids;
}
```

III – Structures de données

Matrice *(matrice.h)*

La structure Matrice se compose de la taille de la matrice (hauteur et largeur) et un pointeur de double vers les données de la matrice.

Pixel & pixDouble *(liste.h & homographie.h)*

La structure pixel fournie les coordonnées (x,y) d'un point sur l'image, en valeur entière.

La structure pixDouble a été rajouté pour permettre les calculs cécessaire à l'anti aliasing, et donne les coordonnées d'un pixel en *double* .

image *(ppm.h)*

La structure *image* contient les données relatives à une image, i.e. un pointeur d'*unsigned char* pour les données de l'image, et les dimensions *width* & *height* de l'image.

RVB *(ppm.h)*

Structure utilisée pour connaître les 3 valeurs RVB d'un pixel, en *unsigned char*.

List *(liste.h)*

Structure permettant de stocker les points de corrélation.

IV – IHM – OpenGL

DESCRIPTION

Nous utilisons OpenGL et Glut pour gérer l'affichage et l'interface utilisateur. Toute la gestion de l'affichage se trouve dans le fichier *gestGlut.c* .

Ce fichier se construit selon les spécificités de glut, il possède, en plus des fonctions tierces, les fonctions suivantes :

Reshape // gère le rafraîchissement de la fenêtre si besoin
Display // gère l'affichage des fenêtres
Keyboard // contrôle du clavier
MouseFunc // contrôle de la souris
Lance (appelé par le main, joue le rôle de fonction principale)

Nous avons cherché à créer une interface aussi développée que possible, avec les connaissances que nous avions. Nos recherches sur internet et l'utilisation du man nous ont également permis d'apprendre un peu plus sur l'utilisation de glut.

Notre programme permet d'afficher les images une fois le calcul de l'homographie effectué.
La touche 's' permet de sauvegarder les images. Ainsi, si l'utilisateur n'est pas satisfait du résultat, aucune image n'est créé sur le disque dur ; il peut à nouveau recommencer les opérations.

DIFFICULTES

Cependant, nous n'avons pas réussi à fermer les fenêtres résultats sans que toutes les autres fenêtres ne se ferment également.

Ceci est du à la notion de contexte OpenGL, signifiant que toutes les fenêtres ouvertes sous un même environnement sont en quelque sorte dépendantes les unes des autres. Lorsque l'on ferme une fenêtre (par la croix en haut à droite par exemple), toues les autres fenêtres de l'environnement de celle ferment également.

La solution viendrait probablement de l'utilisation de fenêtres principales et de sous fenêtres et donc de niveau de hiérarchie. Lors du lancement du programme, une fenêtre principale cachée (glutCreateWindow de la taille de l'écran, glutHideWindow ; puis les fenêtres créées séparément via glutCreateSubWindow) .

V – Manuel d'utilisation

Usage: nom_executable [-h] [-a] [-testInvMat] [-testGauss] image1.ppm image2.ppm image3.ppm
-h ce message.
-a Gestion de l'anti-aliasing.
-testInvMat Teste la fonction d'inversion de matrice avec une matrice aléatoire.
-testGauss Teste la fonction de pivot de Gauss total avec des matrices aléatoires.

Utilisation du programme :

1 : corrélations image1 / image2. Réinitialise les listes\n");
3 : corrélations image3 / image2. Réinitialise les listes\n");
Entrée : pour le panorama 'valdo' uniquement, remplie directement les listes avec des points corrects.
d : Affichage de l'homographie directe
i : Affichage de l'homographie inverse
s : Sauvegarde d'une ou des deux images en fichiers .ppm une fois le calcul effectué

VI – Exemple d'utilisation

L'utilisateur tape sur la ligne de commande :
./mosaique chemin-accès/image1.ppm chemin-accès/image2.ppm chemin-accès/image3.ppm –a

Le programme va charger à l'écran les 3 images qui vont servir à la reconstruction du panoramique.
Il appuie sur '1' pour cliquer la liste de corrélation entre la première et l'image centrale, puis '3' pour l'image centrale et la troisième. Il devra cliquer le même nombre de points entre chaque couple d'image, et un minimum de 4 points de corrélation, sans quoi le programme affichera dans la console ce message d'erreur et réinitialisera les listes (il suffira de taper '1' ou '3' pour reprendre l'exécution du programme).

Pour le panorama 'valdo' uniquement :
Pour éviter de cliquer manuellement les points, la fameuse fonction 'travailAMaPlace' est appelée à l'aide de la touche 'Entrée'. Les 2 listes seront remplies par 4 couples de points judicieusement choisis.

Le calcul et l'affichage des images d'homographie directe et inverse sont gérées séparément. La touche 'd' permet de calculer et d'afficher l'image résultat de l'homographie directe. De la même façon, la touche 'i' permet d'afficher l'image de l'homographie indirecte.

Si le résultat ne convient pas, l'utilisateur peur réinitialiser les listes en appuyant sur la ouche '1' ou '3', et recommencer à cliquer les points, pour relancer ensuite le calcul. Dans ce cas, la ou les images résultats seront cachées, puis réaffichées (et rafraîchies) lors de l'utilisation des touches 'i' ou 'd'.

Les images sont stockées en mémoire pendant l'utilisation du programme. Pour les sauvegarder (ou la sauvegarder s'il n'y en a qu'une de calculée), il suffit d'utiliser la touche 's'.

VII – Difficultés rencontrées

Les principales difficultés lors de ce projet furent la création de la bibliothèque mathématique et les notions de changement de repère.

Le programme se base essentiellement sur des méthodes mathématiques de calculs sur les matrices, notre premier travail consiste à la créer. Le pivot de Gauss total était la fonction la plus dur à implémenter, nous sommes parti sur la création d'une fonction de pivot de Gauss partiel d'abord puis l'avons compléter. Malheureusement, nous n'avons pas trouvé tout de suite comment retenir l'ordre des opération afin de restituer le système d'équation d'origine afin d'effectuer sa résolution.
Cette étape franchie nous devons tester notre bibliothèque, pour cela nous avons tout d'abord comparer avec les résultats donnés par le logiciel Scilab puis avons écrit des fonctions de test directement dans notre programme. Après maintes corrections et de vérification, la bibliothèque fonctionnait correctement.

L'autre difficulté réside dans la recherche de la taille maximale de l'image et le changement de repère dans l'image de résultat.
Les images que nous avons générées au départ sont complètement déformées et sort la plus part du temps du cadre de l'image résultat calculée. Après des discussions avec d'autres élèves, nous nous sommes rendu compte que nous n'avons pas pris en compte le changement de repère pour tous les pixels calculés à partir de l'homographie. La résolution de ce problème fut très rapide.

Par manque de temps, nous n'avons pas pu implémenter toutes les options prévues, notamment la colorimétrie sur les images.

VIII – Compte Rendu

Effectué :

Nous avons mené à bien le projet 'Mosaïque' dans son ensemble.
Ont été réalisées les différentes tâches :

- pivot de Gauss total, gérant les matrices (au lieu de simples vecteurs) ;
- résolution de systèmes surdéterminés ;
- homographie directe ;
- homographie inverse ;
- Création des images résultantes à la volée & enregistrement sur le disque ;
- Anti-aliasing en option.
- Des tests de la bibliothèque mathématique (fonction inversion de matrice et pivot de Gauss total)

Inachevé :

La gestion de la colorimétrie n'a pas été implémentée, faute de temps. Nous en présenterons ici les principes :

La projection d'une image sur une autre n'est pas parfaite : on remarque le contour de l'image superposée à une autre. Ceci est du à la présence d'un léger halo aux extrémités des photographies. La colorimétrie vient corriger ce défaut.

Son principe de base est de calculer une moyenne entre la couleur du pixel de l'image de base et la couleur du pixel à projeter par-dessus.

Chaque image se voit attribuer un poids pour chaque pixel qui la compose allant de 0 à 1. Le pixel du centre de l'image vaut 1, celui à son extrémité vaut 0. Cela donne l'aspect d'u cercle concentrique autour du centre de l'image.

On parcourt donc les parties communes de l'image de base et de celle à projeter. Pour chaque pixel, on applique la formule :

$$e^{-x^2/sigma^2} \quad \text{de même pour :} \; e^{-y^2/sigma^2}$$

Où x et y sont les coordonnées du pixel courant et sigma se calculant par la formule :

$$Sigma = \sqrt{(-L^2/4\ln D)}$$

Où L est la largeur de l'image (ou sa hauteur dans le cas du calcul avec y), et D la distance entre le centre de l'image et le pixel courant.

Pour connaître la valeur du pixel, il suffit de faire un rapport entre les coefficients des 2 pixels obtenus pour obtenir 1 en les additionnant.

Par exemple, si on obtient une valeur de 0.3 pour le pixel de la première image, et 0.5 pour le second :

$$RVB_{final} = (0.3/0.8)* RVB_{img1} + (0.5/0.8)*RVB_{img2}$$

CONCLUSION

Ce projet nous a permis d'approcher des algorithmes de transformation d'image par l'utilisation de calculs matriciels et du pivot de Gauss, ainsi que de nous familiariser avec l'interface Glut/OpenGL.

Ce fut donc un projet concret, qu'il est possible de replacer dans un contexte économique.
En effet, la création d'images panoramiques est un marché à part entière ouvert depuis quelques années.
Ces images nous sont devenues ordinaires sans que nous nous demandions quel type d'appareil pouvait produire ces images à angle de vue si large.
Des appareils existent, mais ils coûtent chers. Les algorithmes tels que celui que nous venons de programmer permettent de produire des panorama tout à fait crédibles avec des appareils photo à angle de vue "basique", d'autant plus que les méthodes que nous avons découvertes sont certainement les bases d'algorithmes plus poussés.

Des applications commerciales de ce type d'image se sont rapidement développées, sur internet notamment. Les hôtels et agences de voyages, agences immobilières y trouvèrent un intérêt certain, grâce aux images à 360°.
Une nouvelle évolution est en passe de se démocratiser grâce aux logiciels de type QuickTime VR d'Apple et consorts, permettant de regarder dans toutes les directions à partir d'un point fixe.
L'évolution du programme vers un panorama à 360°, et a fortiori vers un logiciel type QuickTime VR, est cependant beaucoup plus complexe.

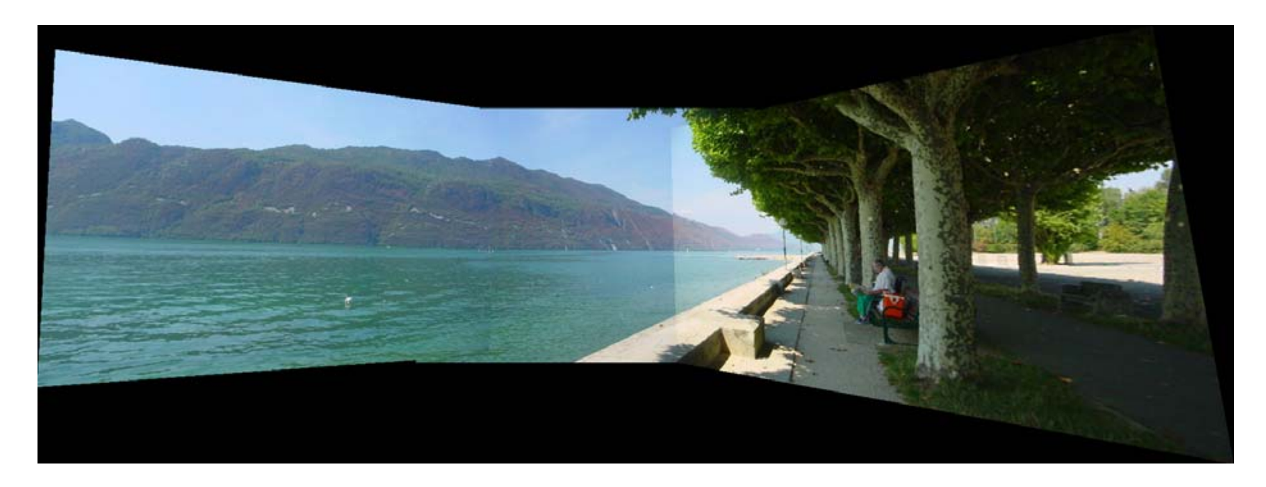
ANNEXES

Voici quelques exemples de mosaïque obtenue en utilisant notre programme :

<u>Exemple 1 :</u>



Image Résultat :



Exemple 2 :



Image Résultat :



Exemple 3 :



Image Résultat :

